18/10/2023
Wednesday

* **Stack and Heap Memory →**

Whenever we create a program, it is assigned with small amount of stack memory and a large amount of heap memory.

for example →

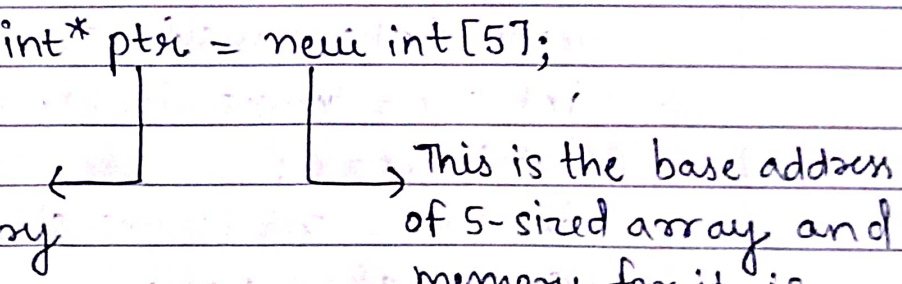
When we write like this → `int arr[5];`

This is a fixed size or static array. That's why memory is allocated to it at compile time or it takes stack memory.

When we write like this → `int* ptr = new int[5];`

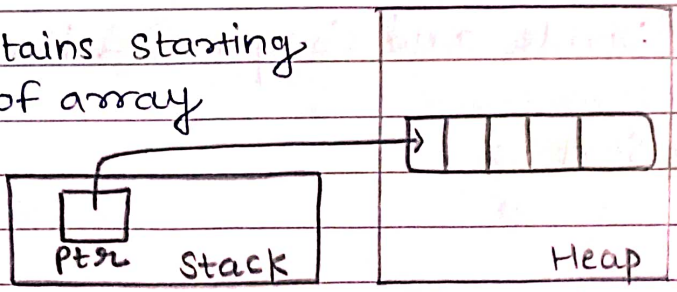
This is a dynamic array. Memory is allocated during run-time and it takes heap memory.

`new` keyword or operator is used if we want dynamic allocation of memory.



So, we can say, a pointer ptr in stack memory is pointing towards the base address of array which is in heap.

ptr contains starting address of array



Static allocation

Dynamic allocation

- We can create new int, new float, new char, new string, new array, or whatever we want to create dynamically.

Note → Jb bhi new keyword ka use krke memory allocation karvate hai to vo return kar raha hota hai starting address uska jiske liye bhi memory allocation karvaya hai.

Note → Always remember to de-allocate memory which is allocated dynamically using delete keyword.

delete[] ptr → for arrays, use this syntax.

DnC (Divide and Conquer) & Backtracking

* **Merge Sort →**

Algorithm →

Step 1 : Break array from mid (S+(e-S)/2). We will get one left part and right part.

Step 2 : Make a recursive call for left part of array. Left part is from S → mid.

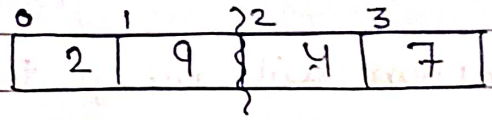
Step 3 : Make a recursive call for right part of array. Right part is from mid+1 → end.

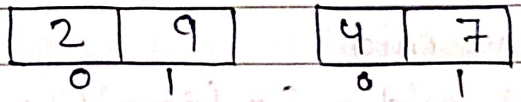
Step 4 : Now, merge these two sorted arrays.

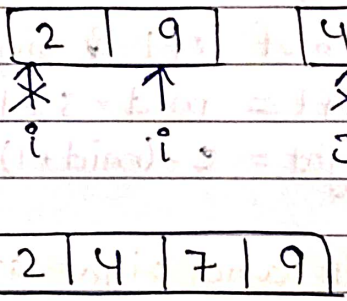
```
mergeSort(arr, s, e)
{
    // Base case
1. Break into left & right.
2. Recursive call for
   left and right.
3. Merge sorted left
   and right part.
}
```

```
merge(arr, s, e)
{
1. Create left and right
   array.
2. Copy values from actual
   array into left and
   right array.
3. Actual merge of 2 sorted
   arrays using 2 pointers.
}
```

Merge logic →

Break → 

Copy left and right array → 

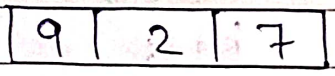
Merge these two sorted arrays using two pointers → 

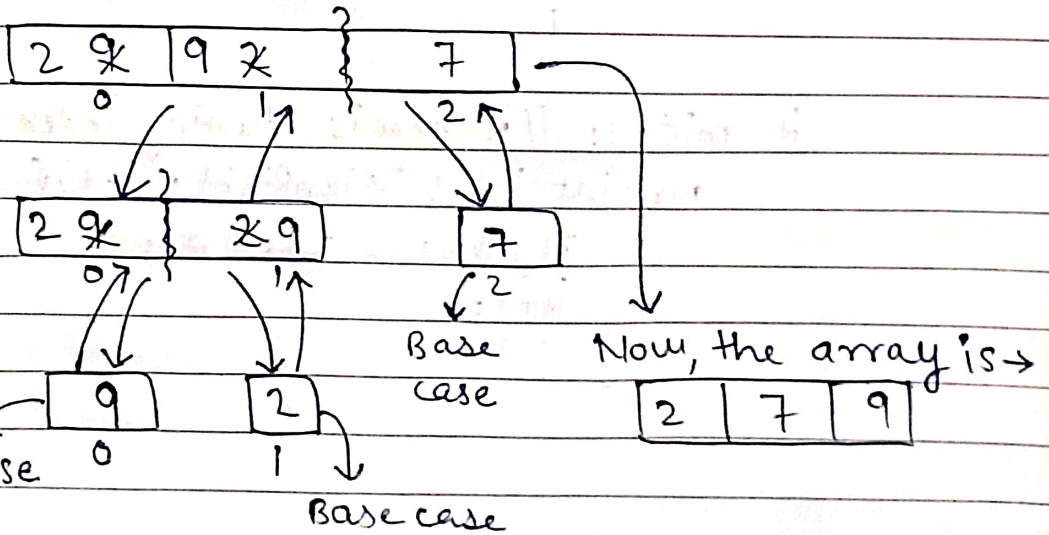
2 < 4
i++

9 > 4
J++

9 > 7
J++

= If one of the array i.e. left or right is exhausted, then as it is copy the rest elements in left or right to the main array.

Dry Run → Let the array 



__/__/____

Code →

```
#include <bits/stdc++.h>
using namespace std;

// This function will merge two sorted arrays

void merge (int arr[], int s, int e){

    // find mid
    int mid = s+ (e-s)/2;

    // find length of left & right array
    int lenLeft = mid - s +1;
    int lenRight = e -(mid+1) +1;

    // create left and right array
    int* left = new int[lenLeft];
    int* right = new int[lenRight];

    // copying values from actual array to left & right
    int k = s ; // k is starting index of left array
    for (int i=0; i< lenLeft; i++){
        left[i] = arr[k];
        k++;
    }

k = mid+1; // k here is starting index of right array
    for (int i=0; i<lenRight; i++){
        right[i]= arr[k];
        k++;
    }
```

// actual merge logic

```
int leftIndex = 0;
int rightIndex = 0;

// yha par sab galti krte hai, mainArrayIndex ko
// 0 se initialise nhi karna hai
int mainArrayIndex = s;
while (leftIndex < lenLeft && rightIndex < lenRight){
    if (left[leftIndex] < right[rightIndex]) {
        arr[mainArrayIndex] = left[leftIndex];
        mainArrayIndex++;
        leftIndex++;
    }
    else {
        arr[mainArrayIndex] = right[rightIndex];
        mainArrayIndex++;
        rightIndex++;
    }
}
// 2 more cases
// case 1 -> Jb left array exhausted & right m elements bache h
    while (rightIndex < lenRight){
        arr[mainArrayIndex] = right[rightIndex];
        mainArrayIndex++;
        rightIndex++;
    }
// Case 2 -> Jb right array exhausted & left m elements bache h
    while (leftIndex < lenLeft){
        arr[mainArrayIndex] = left[leftIndex];
        mainArrayIndex++;
        leftIndex++;
    }
```

```
// de-allocate heap memory
    delete [] left;
    delete [] right;

// This function will recursively divide the array
// into left and right.

void mergeSort (int arr[], int s, int e){
    // base case
    if (s>=e) {
// for empty array & single element case, nothing to
// do, just return.
        return;
    }

    // break
    int mid= s+(e-s)/2;

// recursive call for left array->
    mergeSort (arr, s, mid);

// recursive call for right array->
    mergeSort (arr, mid+1, e);

// merge the sorted arrays
    merge (arr, s, e);
```

```
int main(){
    int arr[]={2,1,3,4,7,6,5};
    int size=7;
    int s=0;
    int e=size-1;

    cout<<"beforemerge sort:"<<endl;
    for(auto num:arr){
        cout<<num<<" ";
    }
    cout<<endl;

    mergeSort(arr,s,e);

    cout<<"after merge Sort:"<<endl;
    for(auto num;arr){
        cout<<nium<<" ";
    }
    cout<<endl;
}
```

I/P = arr = {2, 1, 3, 4, 7, 6, 5}

O/P =
before merge sort:
2 1 3 4 7 6 5
after merge sort:
1 2 3 4 5 6 7

Time complexity of merge sort →

Constant time for Base case

$$T(n) = k_1 + T(n/2) + T(n/2) + n * k$$

- $T(n/2)$ → Time for left half array
- $T(n/2)$ → Time for right half array
- $n * k$ → Time for merge logic

$$T(n) = k_1 + 2T(n/2) + n * k$$

$$2T\left(\frac{n}{2}\right) = 2k_1 + 2 * 2T(n/2^2) + \frac{n}{2} * k * 2$$

$$4T\left(\frac{n}{4}\right) = 4k_1 + 8T\left(\frac{n}{8}\right) + \frac{n}{4} * k * 4$$

$$8T\left(\frac{n}{8}\right) = 8k_1 + 16T\left(\frac{n}{16}\right) + \frac{n}{8} * k * 8$$

$I(+) = k_1$

$$T(n) = k_1(1+2+4+\cdots+2^{a-1}) + (a-1) * n * k$$

$$T(n) = nk_1 + a * n * k$$

$T(n) = n + n\log n$ → Considering the bigger one

$$\boxed{T(n) = O(n\log n)}$$

Space complexity → $O(1)$ = constant space for base case

$O(\log n)$ = Depth of call stack for left half array

$O(\log n)$ = Depth of call stack for right half array

$O(n)$ = space consumed in merge logic

So, $O(1) + O(\log n) + O(\log n) + O(n)$ → Considering bigger one.

So, $\boxed{S.C. = O(n)}$